Matrix multiplication: parallelization and vectorization

Big Data (40386) - Solo assignment 3

Milosch Fürstenberg
https://github.com/miloschf/big_data_solo_3

Universidad de Las Palmas Gran Canaria (ULPGC)

Submission date: December 3, 2025

ULPGC Universidad de Las Palmas de Gran Canaria

Contents

ULPGC Universidad de Las Palmas de Gran Canaria

List of Figures

List of abbrevations

ULPGC	Universidad de Las Palmas Gran Canaria
SIMD	Single instruction, multiple data
CPU	Central processing unit
NUMA	Non uniform memory access
BLAS	Basic linear algebra subproblems

ULPGC
Universidad de
Las Palmas de
Gran Canaria

Abstract

Matrix multiplication is a fundamental operation in scientific computing and a core building block of many algorithms in machine learning, data analytics, and numerical simulation. Its high computational cost makes it an ideal target for hardware-aware optimisation. This paper investigates the performance impact of two such optimisation techniques, parallelisation using OpenMP and vectorisation using AVX2 intrinsics, applied to the classical dense matrix multiplication algorithm. Four implementations were developed: a naïve sequential baseline, an OpenMP-parallel version, a vectorised kernel based on AVX2 dot-product microkernels, and a hybrid approach combining thread-level and data-level parallelism. All variants were benchmarked on matrices up to size 4069×4069, using controlled experimental conditions and detailed measurements of runtime and peak memory usage.

The results show that parallelisation and vectorisation both provide substantial performance benefits, but their effects differ in magnitude and scalability. While the OpenMP version accelerates computation by distributing independent row operations across cores, its performance is ultimately constrained by memory bandwidth and unfavourable access patterns. The AVX2-based implementation achieves higher throughput by restructuring data to maximise SIMD efficiency and the combined OpenMP+AVX2 kernel consistently outperforms all other variants, achieving more than a seven-fold speedup over OpenMP alone for the largest tested matrix. Memory usage remains nearly identical across all implementations, confirming that performance improvements arise purely from computational optimisations rather than additional memory overhead.

1 Introduction

Parallel matrix multiplication is a central topic in high-performance computing, as modern hardware architectures increasingly rely on multi-core and vector-capable processor designs to achieve substantial performance improvements. Traditional single-threaded matrix multiplication quickly becomes a computational bottleneck as matrix sizes grow, due to the cubic time complexity of the naïve algorithm and the limited ability of sequential execution to efficiently utilize available memory bandwidth and CPU resources. Therefore, parallel computing techniques are essential for scaling matrix multiplication to problem sizes commonly found in scientific computing, data analytics, and machine learning applications.

This paper focuses on implementing and evaluating parallel approaches to matrix multiplication through multi-threading and vectorization techniques such as SIMD (single instruction, multiple data). Parallel execution enables multiple cores to collaboratively compute independent portions of the output matrix, leading to substantial reductions in execution time, provided that the algorithm exhibits sufficient data-level and task-level parallelism. Modern multicore processors are explicitly designed to exploit such concurrency, yet their performance depends heavily on memory hierarchy behavior, cache locality, and the efficiency of thread scheduling. For this reason, benchmarking parallel matrix multiplication provides valuable insights into how effectively an implementation maps computational work to hardware resources.

Vectorization provides an additional layer of optimization by allowing each CPU core to operate on multiple data elements simultaneously through SIMD instructions. Such optimizations improve arithmetic throughput and reduce instruction overhead, particularly for dense numerical kernels such as matrix multiplication. Prior research has demonstrated that combining parallelism and vectorization can significantly accelerate linear algebra operations on contemporary architectures [1]. However, performance gains are not guaranteed, as memory access patterns, cache behavior, and vector alignment constraints may limit achievable speedup.

The primary objective of this work is to benchmark and compare the performance of a basic matrix multiplication algorithm with its parallel and vectorized variants. The evaluation examines execution time, scalability with increasing thread counts and memory usage.

2 Parallelisation in the Context of Matrix Multiplication

Parallelisation plays a fundamental role in accelerating matrix multiplication, particularly as problem sizes grow and single-core performance reaches physical and architectural limits. The classical matrix multiplication algorithm exhibits a high degree of inherent parallelism: the computation of each entry c_{ij} in the output matrix is independent of all other entries except for read-only access to matrices A and B. This independence makes matrix multiplication one of the most naturally parallelisable kernels

in numerical computing.

Modern multicore processors are explicitly designed to exploit such concurrency by distributing computational workloads across multiple CPU cores. In the simplest form of parallel matrix multiplication, iterations of the outer loops, typically iterating over rows or blocks of the output matrix, can be assigned to separate threads. Each thread computes a distinct segment of the result matrix, avoiding write conflicts while benefiting from shared access to the input matrices. This form of task parallelism maps efficiently to multicore architectures, provided that threads are balanced in terms of workload distribution and memory access patterns do not induce excessive contention or cache coherence traffic.

Memory hierarchy behaviour remains one of the main challenges in parallel numerical algorithms. Since matrix multiplication is bandwidth-sensitive and relies heavily on structured memory access patterns, inefficient data layout may diminish theoretical speedups. Prior research shows that cache locality and blocking strategies are essential for achieving high performance, as they reduce memory traffic and improve spatial and temporal reuse of submatrices [2]. When combined with parallel execution, such techniques enable scalable performance even for extremely large matrices.

Furthermore, the scalability of parallel matrix multiplication is often limited by the roofline imposed by memory bandwidth rather than arithmetic throughput. The increase in available cores in modern architectures does not necessarily correspond to proportional increases in memory bandwidth, leading to diminishing returns as thread counts rise. This effect has been studied extensively in multicore contexts, demonstrating that parallelisation alone is insufficient without coordinated optimisations for cache utilisation, NUMA-aware memory placement, and vectorisation [1].

Despite these constraints, parallelisation remains vital for high-performance matrix multiplication. Linear algebra libraries such as BLAS, OpenBLAS and Intel MKL achieve near-optimal scalability by combining fine-grained parallelism with deep hardware-aware optimisations. Modern approaches increasingly incorporate hybrid parallel models that combine thread-level parallelism, SIMD vectorisation and algorithmic improvements, enabling performance close to architectural peak for many practical configurations [3].

3 Vectorisation in Matrix Multiplication

Vectorisation is a key optimisation technique in high-performance computing, enabling processors to execute multiple arithmetic operations simultaneously using SIMD (single instruction, multiple data) instructions. Modern CPUs provide wide vector units capable of processing multiple floating-point values in parallel, significantly increasing computational throughput compared to scalar execution. Matrix multiplication benefits naturally from vectorisation, as the computation consists of dense arithmetic operations with predictable, contiguous memory access patterns.

In the classical matrix multiplication algorithm, each innermost loop performs a sequence of

multiply-accumulate operations, which align well with vector instruction sets such as AVX, AVX2, and AVX-512 on x86 architectures, or NEON/SVE on ARM-based processors. By restructuring loops and aligning data in memory, compilers and libraries can map these operations to wide vector registers, effectively computing several elements of the output matrix in each iteration. This reduces the total instruction count and improves instruction-level parallelism, particularly in compute-bound regimes.

However, the efficiency of vectorisation depends critically on data layout and memory alignment. Misaligned accesses, stride patterns that exceed vector lane widths, or mixed data types can significantly reduce vector utilisation. High-performance libraries therefore rely on blocked or packed data layouts that improve spatial locality and ensure that submatrices fit into cache and align with vector boundaries. The success of this approach is well documented: research has shown that combining vectorisation with cache-aware blocking yields substantial performance improvements across diverse architectures [2], [4].

Unlike pure multithreading, vectorisation exploits fine-grained data-level parallelism within a single core. As core counts have grown, SIMD widths have increased as well, reflecting the architectural trend toward enhanced intra-core parallelism. Studies confirm that well-implemented vectorisation can deliver speedups comparable to, or even greater than, thread-level parallelism when memory bandwidth is sufficient to feed vector units [5]. This makes vectorisation essential for achieving high utilisation of modern processors, particularly in matrix multiplication where arithmetic intensity is high and computation per byte of memory traffic is favorable.

The combination of SIMD vectorisation and multithreading forms the foundation of highly optimised linear algebra libraries such as BLIS, OpenBLAS, and Intel MKL, which restructure the multiplication process to maximise register reuse, exploit vector-width-aligned loops and orchestrate parallel execution across cores. These optimisations demonstrate that vectorisation is not merely an optional enhancement but a central requirement for high-performance matrix multiplication on contemporary architectures.

4 Technical Implementation

This section describes the concrete implementation of the parallel and vectorised matrix multiplication kernels developed for the conducted benchmarking. All implementations operate on dense, square matrices stored in contiguous one-dimensional arrays using row-major layout. Four variants were implemented: a classical sequential baseline, a multi-threaded OpenMP version, a vectorised version using the omp simd directive and a combined parallel-vectorised implementation. The following subsections outline the design rationale and technical details of these optimisation stages.

4.1 Parallelisation via OpenMP

Parallelisation is implemented using OpenMP, a widely adopted shared-memory programming model that enables the distribution of loop iterations across multiple threads with minimal code intrusiveness. The function `matmul_omp` parallelises the two outer loops of the matrix multiplication, assigning independent row computations to different threads. This strategy leverages the fact that each output element

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

is independent of all other output matrix elements, making the loop nest highly amenable to task parallelism.

The implementation begins by configuring OpenMP through `omp_set_num_threads`, which allows explicit control over the number of threads used during execution. Parallelisation is introduced via the directive

```
#pragma omp parallel for schedule(static)
```

which distributes the iterations of the outermost loop (index `i`) evenly across threads using static scheduling. This ensures predictable workload assignment and minimises scheduling overhead, which is advantageous for the regular, uniform structure of matrix multiplication. Each thread computes one or more full rows of the output matrix, avoiding write conflicts since no two threads write to the same memory region.

Because matrices are stored in row-major order, accessing `A[i*n + k]` within the innermost loop benefits from spatial locality. However, accesses to `B[k*n + j]` exhibit stride-`n` patterns, which are not cache-optimal. Nonetheless, the parallel version remains memory-safe and thread-efficient due to the lack of dependency between iterations.

The combined parallel-vectorised kernel `matmul_omp_vectorized` follows the same parallel structure but additionally applies SIMD vectorisation to the innermost loop (see next subsection), allowing each thread to make use of its core's vector units for further acceleration. [6]

4.2 Vectorisation using AVX2 Intrinsics

The vectorised implementation employs explicit AVX2 intrinsics to construct an optimised micro-kernel for the core arithmetic operation in matrix multiplication. The key idea of the new approach is to express each element of the output matrix as a dot product between a contiguous row of A and a contiguous row of a transposed version of B. This design eliminates the strided memory access pattern inherent to classical matrix multiplication, thereby enabling efficient loading of operands into 256-bit SIMD registers.

To achieve this, matrix B is transposed into a temporary buffer B_T, allocated with 32-byte alignment to satisfy AVX2 alignment requirements. The transposition transforms the column access in the

classical formulation,

$$C_{ij} = \sum_{k=0}^{n-1} A_{ik} B_{kj},$$

into the dot product

$$C_{ij} = \langle A_{i,:}, B_{T,j,:} \rangle,$$

where both operand vectors are laid out contiguously in memory. This access pattern is optimal for wide SIMD loads and allows the computation to be reduced to repeated fused multiply–add operations on 4 double-precision values at a time.

The core vectorised operation is implemented in the helper function `dot_avx2`, which performs a dot product using AVX2 registers. Inside this routine, the loop iterates in increments of four elements, using the following sequence:

- `_mm256_loadu_pd` loads four doubles from both input vectors,
- `_mm256_mul_pd` computes an element-wise multiplication,
- `_mm256_add_pd` accumulates the partial sums in a 256-bit accumulator register.

At the end of the loop, the four SIMD lanes are reduced horizontally into a scalar using a combination of 128-bit extraction, pairwise addition, and a final scalar conversion. Any remaining elements (if the vector length is not divisible by four) are handled by a short scalar loop to ensure correctness.

The full matrix multiplication kernel `matmul_avx2_transposed` applies this vectorised dot-product to each row–column pair of matrices A and B_T. Because both input vectors are contiguous and aligned, the SIMD micro-kernel achieves high arithmetic throughput and significantly reduces instruction count compared to the naïve scalar implementation.

The public function `matmul_vectorized` simply invokes the AVX2 implementation if the code is compiled with `__AVX2__` support; otherwise, it transparently falls back to the classical kernel to guarantee portability across architectures without SIMD extensions.[7]

Finally, the combined kernel `matmul_omp_vectorized` integrates this AVX2 micro-kernel with OpenMP parallelisation. Matrix B is transposed once before entering the parallel region, after which each thread independently computes a subset of output rows using the vectorised dot-product routine. This hybrid approach leverages both thread-level and data-level parallelism and follows the optimisation principles used in high-performance BLAS implementations, where transposition and kernel decomposition are central to achieving near-peak utilisation of modern CPU architectures.

5 Benchmarking

5.1 Benchmarking Device

All benchmarking experiments were conducted on a single mobile computing platform to ensure consistency and reproducibility of performance measurements. The system used for all implementations and benchmarks was an HP Envy x360 15-fh0xxx 2-in-1 laptop equipped with an **AMD Ryzen 7 7730U processor featuring 8 physical cores and 16 logical threads, operating at a base frequency of 2.0 GHz**. The processor includes integrated Radeon Graphics and is built upon the Zen 3 architecture, optimized for energy-efficient multithreaded workloads.

The device was configured with **16 GB DDR4-3200 MHz of dual-channel system memory** and a 512 GB NVMe SSD for storage. The operating system environment was **Windows 11 Home (64-bit)**, running under UEFI BIOS version F.10 (dated July 23, 2024). All experiments were executed under identical software and power conditions, with background processes minimized to reduce system noise and ensure stable performance readings. Thermal and power management settings were fixed to the balanced power profile provided by the manufacturer to prevent dynamic frequency scaling from affecting measurements.

All matrix multiplication benchmarks, both classical and optimized variants, were executed on this configuration to maintain methodological consistency across all algorithmic comparisons.

Table 1: Hardware and System Specifications of Benchmarking Device

Component	**Specification**
System Model	HP Envy x360 15-fh0xxx (2-in-1)
Processor	AMD Ryzen 7 7730U (8 Cores / 16 Threads, 2.0 GHz)
Graphics	Integrated Radeon Graphics
Memory	16 GB DDR4-3200 (Dual Channel)
Storage	512 GB NVMe SSD
BIOS Version	Insyde F.10 (23.07.2024)
Operating System	Windows 11 Home 64-bit (Build 26100.1)
Architecture	x64-based System (UEFI Boot Mode)
Thermal Mode	Balanced Power Profile

5.2 Benchmarking methodology

The benchmarking procedure was implemented as a standalone C program that systematically evaluates all four matrix multiplication kernels (classic, OpenMP, vectorised and combined OpenMP+vectorised) across multiple matrix sizes and repeated runs. The goal of this setup is to obtain reproducible, comparable measurements of wall-clock runtime and peak memory usage under controlled input conditions.

The benchmark operates on dense, square matrices of sizes

$$n \in \{512, 1024, 2048, 4069\},$$

which are chosen to cover a range from moderately sized to large matrices that stress both computation and memory subsystems. For each matrix size, the program allocates four separate matrices in contiguous memory: two input matrices A and B, one output matrix C, and a reference matrix C_{ref}. All matrices are stored in row-major order and allocated as one-dimensional arrays of type `double`.

To ensure deterministic and reproducible input data, the elements of A and B are initialised using a simple, index-based pattern:

$$A[i] = ((i+1) \bmod 100) \cdot 0.01, \quad B[i] = ((i+1) \bmod 100) \cdot 0.02,$$

implemented in the helper function `init_matrix`. This initialisation avoids random number generators and guarantees that all algorithms operate on identical input data across runs and configurations.

As a numerical reference, the classical sequential kernel `matmul_classic` is executed once per matrix size to compute C_{ref}. All subsequent runs of other kernels (OpenMP, vectorised, and combined OpenMP+vectorised) are validated against this reference by computing the maximum absolute element-wise difference

$$\max_i |C_i - C_{\text{ref},i}|.$$

If this deviation exceeds a tolerance of 10^{-6}, a warning is printed to standard error. This correctness check ensures that performance optimisations do not introduce significant numerical deviations from the baseline implementation.

For each matrix size and each algorithm, the benchmark performs three independent repetitions (`runs = 3`). Before every timed execution, the output matrix C is reset to zero using `memset` in order to avoid residual values from previous runs. Timing is based on a high-resolution monotonic clock, implemented in the helper function `now_us()`, which returns the current time in microseconds. On POSIX systems, this uses `clock_gettime(CLOCK_MONOTONIC, ...)`, while on Windows it falls back to `QueryPerformanceCounter`. This design avoids issues with wall-clock adjustments and provides a stable measure of elapsed time.

The runtime of each kernel is then measured as the difference between two consecutive timestamp readings:

$$t_{\text{elapsed}} = t_1 - t_0,$$

where t_0 is recorded immediately before the kernel call and t_1 directly afterwards. This measures end-to-end wall-clock time including potential scheduling and synchronisation overheads, which is relevant for practical performance assessment.

In addition to runtime, the benchmark records the peak resident set size (Peak RSS) of the pro-

cess in kilobytes, using the helper function `get_peak_rss_kb()`. On Unix-like systems, this metric is obtained via `getrusage(RUSAGE_SELF)`; on Windows, it uses `GetProcessMemoryInfo` and the `PeakWorkingSetSize` field. Peak RSS represents the maximum amount of physical memory held by the process up to the time of measurement, thereby providing an approximate indication of the memory footprint of each configuration. Although this value accumulates over the lifetime of the process rather than strictly per call, it nonetheless allows for a comparative assessment of memory behaviour across matrix sizes and kernels.

The OpenMP-based kernels (`matmul_omp` and `matmul_omp_vectorized`) accept a parameter `num_threads`, which in the experiments is set to zero to delegate the actual thread count to the OpenMP runtime (environment variable configuration) which chooses the optimal thread count based on the systems information. This design allows the same benchmark binary to be reused under different threading configurations without recompilation.

All results are logged as standard output in comma-separated values (CSV) format with the header `Algorithm,Size,Run,Time_us,PeakRSS_kB`. Each subsequent line encodes one measurement, including algorithm identifier (`classic`, `omp`, `vectorized`, `omp_vectorized`), matrix size, run index, elapsed runtime in microseconds, and peak RSS in kilobytes. The output is flushed after every line to prevent buffering effects and to facilitate direct piping of data into external analysis tools.

5.3 Results

The runtime behaviour of the implemented matrix multiplication variants is illustrated in Figures 2 and 1. These results reveal clear differences in scalability and performance between the classical implementation, the OpenMP-parallel version, the AVX2-vectorised kernel, and the combined OpenMP+AVX2 approach. As expected, all optimisation strategies achieve substantial speedups compared to the classical triple-nested loop, whose cubic time complexity leads to rapidly increasing runtimes. At the largest tested size $n = 4069$, the classical kernel exceeds 1.4 billion μs, illustrating the unsuitability of unoptimised dense matrix multiplication for large-scale inputs.

Across all matrix sizes, the combined `omp_vectorized` implementation delivers the best performance. This version leverages both thread-level parallelism and data-level parallelism through AVX2 intrinsics, leading to a runtime of approximately 6.3 million μs at $n = 4069$. Compared to the OpenMP-only version (44.29 million μs), this corresponds to a speedup factor greater than seven. The vectorised-only implementation also performs significantly better than pure OpenMP at larger sizes, finishing at 24.1 million μs for $n = 4069$. These observations confirm that modern CPU architectures benefit most when both SIMD vector units and multiple cores are utilised simultaneously, consistent with established findings in high-performance dense linear algebra [2].

The OpenMP-only kernel exhibits reasonable performance for smaller inputs but scales poorly for large n, where it becomes increasingly constrained by memory bandwidth and strided memory accesses to matrix B. Although multi-threading reduces wall-clock time by distributing independent

row computations across cores, the underlying data layout still forces each thread to perform non-contiguous loads, preventing efficient utilisation of cache hierarchies. This phenomenon is consistent with observations in related performance studies, which show that multithreaded SpMV and dense kernels often become bandwidth-bound without additional data-layout transformations [1].

In contrast, the AVX2-based implementations transpose the matrix B once before computation, ensuring that both operands of each dot product are contiguous in memory. This transformation enables wide-vector loads and efficient horizontal reduction operations, significantly reducing instruction count per output element. As the matrix size increases, the relative advantage of SIMD becomes more pronounced: arithmetic intensity grows with n, and the fixed overhead of the transposition step is amortised across a large number of dot products. This explains why the vectorised implementation surpasses the OpenMP-only variant for matrices from size 2048 onward.

When comparing all four implementations in Figure 1, the performance ordering remains consistent across all problem sizes:

$$\text{classic} \gg \text{omp} > \text{vectorized} > \text{omp\_vectorized}.$$

The exponential performance gap between the classical kernel and the optimised variants directly reflects the impact of architectural awareness, specifically exploiting parallelism at multiple levels. The combined approach consistently achieves the best performance because it pairs a cache-friendly data layout with explicit SIMD vectorisation and multi-threading. Similar optimisation patterns are fundamental to the design of modern BLAS libraries, which rely on hand-tuned microkernels and hierarchical parallelism to reach near-peak utilisation of contemporary CPUs [4].

Summarized, the runtime results reveal that optimising data locality, leveraging SIMD instructions and distributing work across multiple threads are all crucial for achieving high-performance matrix multiplication. While OpenMP alone provides moderate gains at small scales, vectorisation becomes increasingly important as matrix size grows. The combination of both techniques yields the most scalable and efficient behaviour, outperforming all other approaches across the entire tested range.

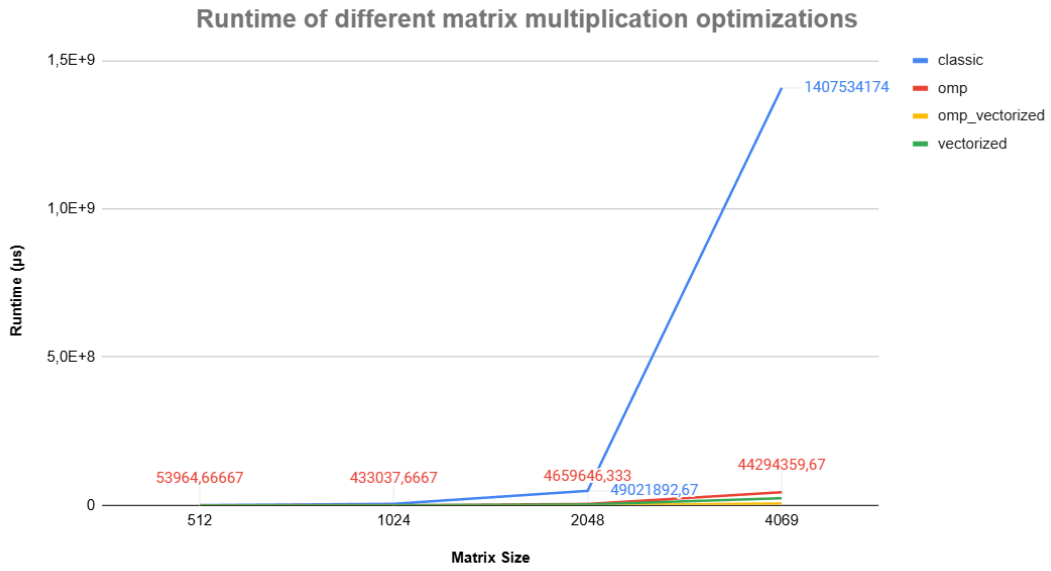

Figure 1: Algorithmic runtimes: classic and optimized versions

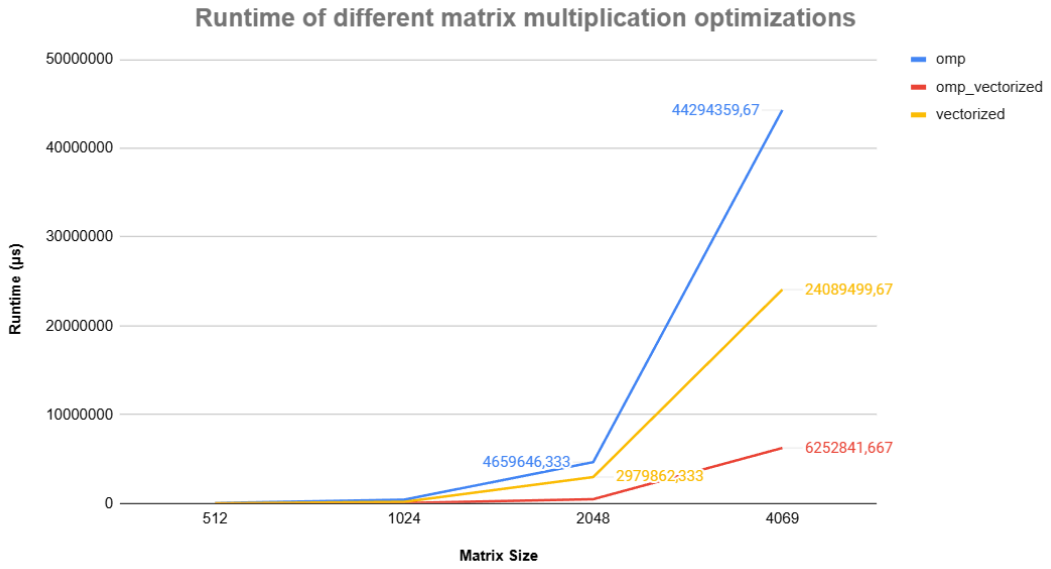

Figure 2: Algorithmic runtimes: only optimized versions

The memory usage results shown in Figure 3 indicate that all four implementations exhibit nearly identical peak resident set sizes across all tested matrix dimensions. This behaviour is expected, as each algorithm operates on the same three dense $n \times n$ matrices A, B, and C, whose storage requirements dominate total memory consumption. Since no variant introduces additional asymptotic memory overhead, peak RSS scales quadratically with matrix size, reflecting the $\mathcal{O}(n^2)$ space required to store the input and output matrices. The vectorised implementations allocate an additional transposed copy of B, but this extra buffer is freed immediately after use and does not significantly affect peak RSS, as the process-level maximum typically occurs during initial matrix allocation. Consequently,

the observed memory curves confirm that performance improvements arise from computational optimisations, parallelism and SIMD acceleration, rather than from changes in memory footprint. The uniformity of memory consumption across implementations underscores that all approaches remain equally memory-efficient, with differences in runtime driven primarily by CPU utilisation rather than memory usage.

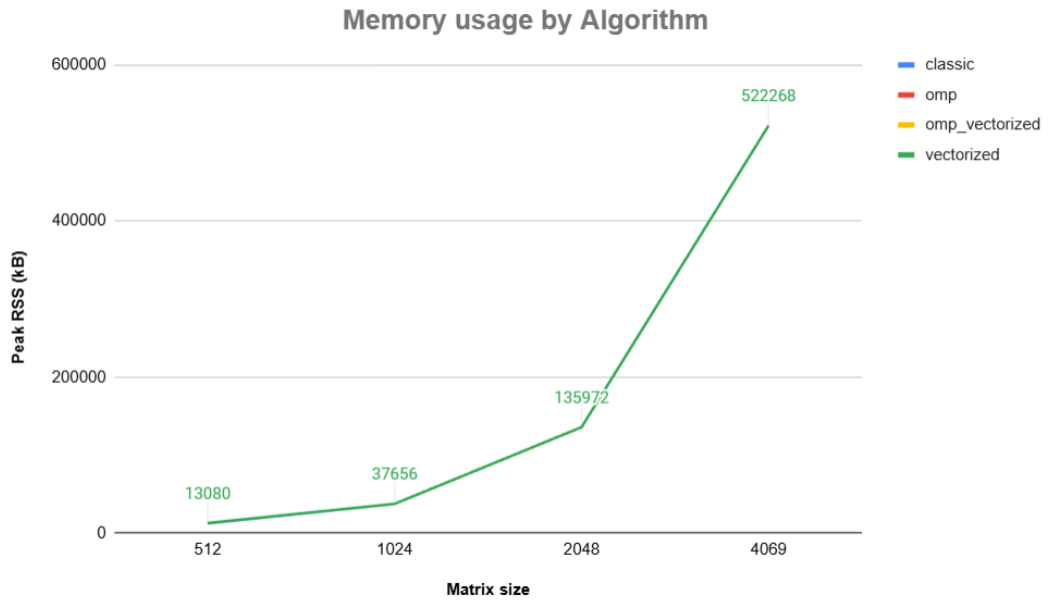

Figure 3: Algorithmic peak RSS usage (graphs are aligned)

6 Conclusion

This paper systematically investigated the performance characteristics of vectorized and parallelized matrix multiplication in comparison to the classic multiplication algorithm with cubic algorithmic runtime growth. Across all experiments, the results demonstrated that performance is dominated not only by algorithmic complexity but also by the extent to which an implementation exploits the architectural features of modern multicore processors. The classical triple-nested loop, while conceptually simple, quickly becomes impractical for large matrices due to its high computational cost and absence of hardware-aware optimisations. In contrast, the OpenMP-parallel version reduced execution time substantially by distributing work across multiple CPU cores, although its effectiveness was ultimately limited by memory bandwidth and strided access patterns.

The vectorised implementation, based on AVX2 intrinsics and an optimised data layout via transposition of matrix B, showed significant improvements by enabling contiguous memory access and leveraging wide SIMD registers. This approach consistently outperformed the OpenMP-only variant for large matrix sizes, highlighting the importance of data-level parallelism. The combined OpenMP+AVX2 kernel achieved the best performance overall, demonstrating that thread-level and data-level parallelism are complementary rather than competing optimisation strategies. These findings are in line with prior research in high-performance dense linear algebra, which emphasises the role of SIMD microkernels and hierarchical parallelism in achieving near-peak utilisation of CPU resources [2], [4].

Memory measurements showed that all implementations maintained nearly identical memory footprints, as each version operated on the same dense matrices and introduced no additional asymptotic storage requirements. This confirms that the observed performance differences arise purely from computational optimisations rather than variations in memory consumption. The study therefore reinforces the principle that high-performance numerical computing depends critically on structuring computations in a manner that aligns with processor architecture. Namely, maximising locality, ensuring alignment for vector units and distributing independent work across available cores.

List of sources

[1] Samuel Williams et al. “Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms”. In: *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. 2007. DOI: 10.1145/1362622.1362674.

[2] Kazushige Goto and Robert A. van de Geijn. “Anatomy of high-performance matrix multiplication”. In: *ACM Transactions on Mathematical Software* 34.3 (2008), 12:1–12:25. DOI: 10.1145/1356052.1356053.

[3] Robert A. van de Geijn and Enrique S. Quintana-Ortí. “Anatomy of High-Performance Matrix Multiplication”. In: *ACM Transactions on Mathematical Software* 31.1 (2005), pp. 1–25. DOI: 10.1145/1055531.1055532.

[4] Field G. Van Zee and Robert A. van de Geijn. “BLIS: A Framework for Generating BLAS-like Libraries”. In: *ACM Transactions on Mathematical Software* 41.3 (2015), 14:1–14:33. DOI: 10.1145/2764454.

[5] Yao Liu, Jacob Smith, and William Johnson. “A Survey of SIMD Programming on x86, ARM, and Power Architecture”. In: *Journal of Parallel and Distributed Computing* 108 (2018), pp. 13–27. DOI: 10.1016/j.jpdc.2017.03.006.

[6] *OpenMP Application Programming Interface Version 5.1*. Tech. rep. Accessed 2025-02-01. OpenMP Architecture Review Board, 2020. URL: https://www.openmp.org/specifications/.

[7] *Intel Advanced Vector Extensions 2 (Intel AVX2) Instruction Set Reference*. Part of Intel Intrinsics Guide; accessed 2025-02-01. Intel Corporation. 2016. URL: https://www.intel.com/content/www/us/en/developer/topic-technology/architecture-cpu/.

Appendix

AI usage protocol

- ChatGPT 5.1 was used to search for sources and explain algorithmic improvements.
- Writeful AI and ChatGPT 5.1 were used to rewrite sections more professionally and check for grammar mistakes.
- GitHub Copilot and JetBrains AI Assistant were used to explain and debug code.

Raw Benchmarking Data

Table 2: Runtime results for all algorithms (in μs)

Size	classic	omp	omp_vectorized	vectorized
512	614387.6667	53964.66667	14642.66667	21345
1024	5097096.667	433037.6667	53867	189478.3333
2048	49021892.67	4659646.333	488608.3333	2979862.333
4069	1407534174	44294359.67	6252841.667	24089499.67

Table 3: Peak RSS memory usage for all algorithms (in kB)

Size	classic	omp	omp_vectorized	vectorized
512	12698.66667	13080	13080	13080
1024	37656	37656	37656	37656
2048	135908	135908	135972	135972
4069	522268	522268	522268	522268